Towards Correctness of Program Transformations Through Unification and Critical Pair Computation

Conrad Rau[*] and Manfred Schmidt-Schauß

Institut für Informatik
Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt, Germany

{rau,schauss}@ki.informatik.uni-frankfurt.de

Correctness of program transformations in extended lambda calculi with a contextual semantics is usually based on reasoning about the operational semantics which is a rewrite semantics. A successful approach to proving correctness is the combination of a context lemma with the computation of overlaps between program transformations and the reduction rules, and then of so-called complete sets of diagrams. The method is similar to the computation of critical pairs for the completion of term rewriting systems. We explore cases where the computation of these overlaps can be done in a first order way by variants of critical pair computation that use unification algorithms. As a case study we apply the method to a lambda calculus with recursive let-expressions and describe an effective unification algorithm to determine all overlaps of a set of transformations with all reduction rules. The unification algorithm employs many-sorted terms, the equational theory of left-commutativity modelling multi-sets, context variables of different kinds and a mechanism for compactly representing binding chains in recursive let-expressions.

1 Introduction and Motivation

Programming languages are often described by their syntax and their operational semantics, which in principle enables the implementation of an interpreter and a compiler in order to put the language into use. Of course, also optimizations and transformations into low-level constructs are part of the implementation. The justification of correctness is in many cases either omitted, informal or by intuitive reasoning. Inherent obstacles are that programming languages are usually complex, use operational features that are not deterministic like parallel execution, concurrent threads, and effects like input and output, and may even be modified or extended in later releases.

Here we want to pursue the approach using contextual semantics for justifying the correctness of optimizations and compilation and to look for methods for automating the correctness proofs of transformations and optimizations.

We assume given the syntax of programs $\mathcal{P}$, a deterministic reduction relation $\rightarrow \subseteq \mathcal{P} \times \mathcal{P}$ that represents a single execution step on programs

and values that represent the successful end of program execution. The reduction of a program may be non-terminating due to language constructs that allow iteration or recursive definitions. For a program $P \in \mathcal{P}$ we write $P\Downarrow$ if there is a sequence of reductions to a value, and say P *converges* (or terminates successfully) in this case. Then equivalence of programs can be defined by $P_1 \sim P_2 \iff \big(\text{for all } C : C[P_1]\Downarrow \iff C[P_2]\Downarrow\big)$, where C is a context, i.e. a program with a hole $[\cdot]$ at a single position. Justifying the correctness of a program transformation $P \rightsquigarrow P'$ means to provide a proof that $P \sim P'$. Unfortunately,

[*]This author is supported by the DFG under grant SCHM 986/9-1.

the quantification is over an infinite set: the set of all contexts, and the criterion is termination, which is undecidable in general. Well-known tools to ease the proofs are context lemmas [9], ciu-lemmas [6] and bisimulation, see e.g. [7].

The reduction relation $\rightarrow$ is often given as a set of rules $l_i \rightarrow r_i$ similarly to rewriting rules, but extended with different kinds of meta-variables and some other constructs, together with a strategy determining when to use which rule and at which position. In order to prove correctness of a program transformation that is also given in a rule form $s_1 \rightarrow s_2$, we have to show that $\sigma(s_1) \sim \sigma(s_2)$ for all possible rule instantiations σ i.e. $C[\sigma(s_1)]\Downarrow \iff C[\sigma(s_2)]\Downarrow$ for all contexts C. Using the details of the reduction steps and induction on the length of reductions, the hard part is to look for conflicts between instantiations of s_1 and some l_i, i.e. to compute all the overlaps of l_i and s_1, and the possible completions under reduction and transformation. This method is reminiscent of the critical pair criterion of Knuth-Bendix method [8] but has to be adapted to an asymmetric situation, to extended instantiations and to higher-order terms.

In this paper we develop a unification method to compute all overlaps of left hand sides of a set of transformations rules and the reduction rules of the calculus L_{need} which is a call-by-need lambda calculus with a letrec-construct (see [12]). We show that a custom-tailored unification algorithm can be developed that is decidable and produces a complete and finite set of unifiers for the required equations. The following expressiveness is required: *Many-sorted terms* in order to avoid most of the junk solutions; *context variables* which model the context meta-variables in the rule descriptions; *context classes* allow the unification algorithm to treat different kinds of context meta-variables in the rules; the *equational theory of multi-sets* models the letrec-environment of bindings; *Empty sorts* are used to approximate scoping rules of higher-order terms, where, however, only the renaming can be modeled. Since the reduction rules are linear in the meta-variables, we finally only have to check whether the solutions produce expressions that satisfy the distinct variable convention. *Binding Chains* in letrec-expressions are a syntactic extension that models binding sequences of unknown length in the rules. This also permits to finitely represent infinitely many unifiers, and thus is indispensable for effectively computing all solutions.

The required complete sets of diagrams can be computed from the overlaps by applying directed transformations and reduction rules. These can be used to prove correctness of program transformations by inductive methods.

Since our case study is done for a small calculus, the demand for extending the method to other calculi like the extended lambda calculus in [15] would justify further research.

In Section 2 we present the syntax and operational semantics of a small call-by-need lambda calculus with a cyclic let. The normal order reduction rules and transformations are defined. In Section 3, the translation into extended first-order terms is explained. Section 4 contains a description of the unification algorithm that computes overlaps of left hand sides of rules and transformations in a finite representation. Finally, in Section 5, we illustrate a run of the unification algorithm by an example.

2 A Small Extended Lambda Calculus with letrec

In this section we introduce the syntax and semantics of a small call-by-need lambda calculus and use it as a case-study. Based on the definition of the small-step reduction semantics of the calculus we define our central semantic notion of *contextual equivalence* of calculi expressions and correctness of program transformations. We illustrate a method to prove the correctness of program transformations which uses a *context lemma* and *complete sets of reduction diagrams*.

2.1 The Call-by-Need Calculus L_{need}

We define a simple call-by-need lambda calculus L_{need} which is exactly the call-by-need calculus of [12]. Calculi that are related are in [14], and [1].

The set $\mathcal{E}$ of L_{need}-expressions is as follows where x, x_i are variables:

$$s_i, s, t \in \mathcal{E} \quad ::= \quad x \mid (s\ t) \mid (\lambda x.s) \mid (\texttt{letrec}\, x_1 = s_1, \ldots, x_n = s_n \,\texttt{in}\, t)$$

We assign the names *application*, *abstraction*, or *letrec-expression* to the expressions $(s\ t)$, $(\lambda x.s)$, $(\texttt{letrec}\, x_1 = s_1, \ldots, x_n = s_n \,\texttt{in}\, t)$, respectively. A group of letrec-bindings, also called *environment*, is abbreviated as *Env*.

We assume that variables x_i in letrec-bindings are all distinct, that letrec-expressions are identified up to reordering of binding-components (i.e. the binding-components can be interchanged), and that, for convenience, there is at least one binding. Letrec-bindings are recursive, i.e., the scope of x_j in $(\texttt{letrec}\, x_1 = s_1, \ldots, x_{n-1} = s_{n-1} \,\texttt{in}\, s_n)$ are all expressions s_i with $1 \leq i \leq n$. Free and bound variables in expressions and α-renamings are defined as usual. The set of free variables in t is denoted as $FV(t)$. We use the distinct variable convention (DVC), i.e., all bound variables in expressions are assumed to be distinct, and free variables are distinct from bound variables. The reduction rules are assumed to implicitly α-rename bound variables in the result if necessary.

A *context* C is an expression from L_{need} extended by a symbol $[\cdot]$, the *hole*, such that $[\cdot]$ occurs exactly once (as sub-expression) in C. A formal definition is:

Definition 2.1 Contexts $\mathcal{C}$ *are defined by the following grammar:*

$$C \in \mathcal{C} \quad ::= \quad [\cdot] \mid (C\ s) \mid (s\ C) \mid (\lambda x.C) \mid \ (\texttt{letrec}\, x_1 = s_1, \ldots, x_n = s_n \,\texttt{in}\, C) \mid \ (\texttt{letrec}\, Env, x = C \,\texttt{in}\, s)$$

Given a term t and a context C, we write $C[t]$ for the L_{need}-expression constructed from C by plugging t into the hole, i.e, by replacing $[\cdot]$ in C by t, where this replacement is meant syntactically, i.e., a variable capture is permitted. Note that α-renaming of contexts is restricted.

Definition 2.2 *The* unrestricted reduction rules *for the calculus L_{need} are defined in Figure 1. Several reduction rules are denoted by their name prefix, e.g. the union of (llet-in) and (llet-e) is called (llet), the union of (cp-e) and (cp-in) is called (cp), the union of (llet) and (lapp) is called (lll).*

(lbeta)	$((\lambda x.s)\ r) \rightarrow (\texttt{letrec}\, x = r \,\texttt{in}\, s)$
(cp-in)	$(\texttt{letrec}\, x = s, Env \ \texttt{in}\, C[x]) \rightarrow (\texttt{letrec}\, x = s, Env \ \texttt{in}\, C[s])$
	where s is an abstraction or a variable
(cp-e)	$(\texttt{letrec}\, x = s, Env, y = C[x] \,\texttt{in}\, r) \ \rightarrow (\texttt{letrec}\, x = s, Env, y = C[s] \,\texttt{in}\, r)$
	where s is an abstraction or a variable
(llet-in)	$(\texttt{letrec}\, Env_1 \,\texttt{in}\, (\texttt{letrec}\, Env_2 \,\texttt{in}\, r)) \rightarrow (\texttt{letrec}\, Env_1, Env_2 \,\texttt{in}\, r)$
(llet-e)	$(\texttt{letrec}\, Env_1, x = (\texttt{letrec}\, Env_2 \,\texttt{in}\, s_x) \,\texttt{in}\, r) \rightarrow (\texttt{letrec}\, Env_1, Env_2, x = s_x \,\texttt{in}\, r)$
(lapp)	$((\texttt{letrec}\, Env \,\texttt{in}\, t)\ s) \rightarrow (\texttt{letrec}\, Env \,\texttt{in}\, (t\ s))$

Figure 1: Unrestricted reduction rules of L_{need} (also used as transformations)

The reduction rules of L_{need} contain different kinds of meta-variables. The meta-variables r, s, s_x, t denote arbitrary L_{need}-expressions. Env, Env_1, Env_2 represent letrec-environments and x, y denote bound variables. All meta-variables can be instantiated by an L_{need}-expression of the appropriate syntactical form. A reduction rule $\rho = l \to r$ is applicable to an expression e if l can be matched to e. Note that an expression may contain several sub-expressions that can be reduced according to the reduction rules of Figure 1.

A standardizing order of reduction is the *normal order reduction* (see definitions below) where reduction takes place only inside *reduction contexts*.

Definition 2.3 Reduction contexts $\mathcal{R}$, application contexts $\mathcal{A}$ *and* surface contexts $\mathcal{S}$ *are defined by the following grammars:*

$$\begin{array}{lcl}
A \in \mathcal{A} & := & [\cdot] \mid (A\ s) \qquad \textit{where } s \textit{ is an expression.} \\
R \in \mathcal{R} & := & A \mid \texttt{letrec}\ \mathit{Env}\ \texttt{in}\ A \mid \texttt{letrec}\ y_1 = A_1, \mathit{Env}\ \texttt{in}\ A[y_1] \\
 & & \mid \texttt{letrec}\ y_1 = A_1, \{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^{n}, \mathit{Env}\ \texttt{in}\ A[y_n] \\
S \in \mathcal{S} & := & [\cdot] \mid (S\ s) \mid (s\ S) \mid\ (\texttt{letrec}\ y_1 = s_1, \ldots, y_n = s_n\ \texttt{in}\ S) \mid\ (\texttt{letrec}\ \mathit{Env}, y = S\ \texttt{in}\ s)
\end{array}$$

A sequence of bindings of the form $y_{m+1} = A_{m+1}[y_m], y_{m+2} = A_{m+2}[y_{m+1}], \ldots, y_n = A_n[y_{n-1}]$ where the y_i are distinct variables, the A_i are not the empty context and $m < n$ is called a *binding chain* and abbreviated by $\{y_{i+1} = A_{i+1}[y_i]\}_{i=m}^{n}$.

Definition 2.4 *Normal order reduction* $\xrightarrow{no}$ *(called no-reduction for short) is defined by the reduction rules in Figure 2.*

(lbeta)	$R[(\lambda x.s)\ r] \to R[\texttt{letrec}\ x = r\ \texttt{in}\ s]$
(cp-in)	$\texttt{letrec}\ y = s, \mathit{Env}\ \texttt{in}\ A[y] \to \texttt{letrec}\ y = s, \mathit{Env}\ \texttt{in}\ A[s]$
	where s is an abstraction or a variable.
(cp-e)	$\texttt{letrec}\ y_1 = s, y_2 = A_2[y_1], \mathit{Env}\ \texttt{in}\ A[y_2] \to \texttt{letrec}\ y_1 = s, y_2 = A_2[s], \mathit{Env}\ \texttt{in}\ A[y_2]$
(cp-e-c)	$\texttt{letrec}\ y_1 = s, y_2 = A_2[y_1], \{y_{i+1} = A_{i+1}[y_i]\}_{i=2}^{n}, \mathit{Env}\ \texttt{in}\ A[y_n]$
	$\to \texttt{letrec}\ y_1 = s, y_2 = A_2[s], \{y_{i+1} = A_{i+1}[y_i]\}_{i=2}^{n}, \mathit{Env}\ \texttt{in}\ A[y_n]$
	in the cp-e rules s is an abstraction or a variable and A_2 is a non-empty context.
(llet-in)	$(\texttt{letrec}\ \mathit{Env}_1\ \texttt{in}\ (\texttt{letrec}\ \mathit{Env}_2\ \texttt{in}\ r)) \to (\texttt{letrec}\ \mathit{Env}_1, \mathit{Env}_2\ \texttt{in}\ r)$
(llet-e)	$\texttt{letrec}\ y_1 = (\texttt{letrec}\ \mathit{Env}_1\ \texttt{in}\ r), \mathit{Env}_2\ \texttt{in}\ A[y_1] \to \texttt{letrec}\ y_1 = r, \mathit{Env}_1, \mathit{Env}_2\ \texttt{in}\ A[y_1]$
(llet-e-c)	$\texttt{letrec}\ y_1 = (\texttt{letrec}\ \mathit{Env}_1\ \texttt{in}\ r), \{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^{n}, \mathit{Env}_2\ \texttt{in}\ A[y_n]$
	$\to \texttt{letrec}\ y_1 = r, \mathit{Env}_1, \{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^{n}, \mathit{Env}_2\ \texttt{in}\ A[y_n]$
(lapp)	$R[((\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ r)\ t)] \to R[(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (r\ t))]$

Figure 2: Normal order reduction rules of L_{need}

Note that the normal order reduction is unique. A *weak head normal form in L_{need} (WHNF)* is defined as either an abstraction $\lambda x.s$, or an expression $(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ \lambda x.s)$.

The *transitive closure* of the reduction relation $\to$ is denoted as $\xrightarrow{+}$ and the *transitive and reflexive closure* of $\to$ is denoted as $\xrightarrow{*}$. Respectively we use $\xrightarrow{no,+}$ for the transitive closure of the normal order reduction relation, $\xrightarrow{no,*}$ for its reflexive-transitive closure, and $\xrightarrow{no,k}$ to indicate k normal order reduction steps. If for an expression t there exists a (finite) sequence of normal order reductions $t \xrightarrow{no,*} t'$ to a WHNF t', we say that the reduction *converges* and denote this as $t \Downarrow t'$ or as $t\Downarrow$ if t' is not important. Otherwise the reduction is called *divergent* and we write $t\Uparrow$.

The semantic foundation of our calculus L_{need} is the equality of expressions defined by contextual equivalence.

Definition 2.5 (Contextual Preorder and Equivalence) *Let s,t be L_{need}-expressions. Then:*

$$\begin{array}{lll} s \leq_c t & \textit{iff} & \forall C:\ C[s]\Downarrow \Rightarrow C[t]\Downarrow \\ s \sim_c t & \textit{iff} & s \leq_c t \wedge t \leq_c s \end{array}$$

Definition 2.6 *A* program transformation *$T \subseteq L_{need} \times L_{need}$ is a binary relation on L_{need}-expressions. A program transformation is called* correct *iff $T \subseteq \sim_c$.*

Program transformations are usually given in a format similarly to reduction rules (as in Figure 1 and Figure 2). A program transformation T is written as $s \xrightarrow{T} t$ where s,t are meta-expressions i.e. expression that contain meta-variables. Here we restrict our attention for the sake of simplicity to the program transformations that are given by the reduction rules in Figure 1.

An important tool to prove contextual equivalence is a *context lemma* (see for example [9], [13],[15]), which allows to restrict the class of contexts that have to be considered in the definition of the contextual equivalence from general $\mathcal{C}$ to $\mathcal{R}$ contexts.

However, often $\mathcal{S}$-contexts are more appropriate for computing overlaps and closing the diagrams, so we will use $\mathcal{S}$-contexts instead of $\mathcal{R}$-contexts.

Lemma 2.7 *Let s,t be L_{need}-expressions and S a context of class $\mathcal{S}$. $(S[s]\Downarrow \Rightarrow S[t]\Downarrow)$ iff $\forall C: (C[s]\Downarrow \Rightarrow C[t]\Downarrow)$; i.e. $s \leq_c t$.*

Proof. A proof of this lemma when the contexts are in class $\mathcal{R}$ is in [13]. Since every $\mathcal{R}$-context is also an $\mathcal{S}$-context, the lemma holds. □

To prove the correctness of a transformation $s \xrightarrow{T} t$ we have to prove that $s \sim_c t \Leftrightarrow s \leq_c t \ \wedge\ t \leq_c s$ which by Definition 2.5 amounts to showing $\forall C: C[s]\Downarrow \Rightarrow C[t]\Downarrow \ \wedge\ C[t]\Downarrow \Rightarrow C[s]\Downarrow$. The context lemma yields that it is sufficient to show $\forall S: S[s]\Downarrow \Rightarrow S[t]\Downarrow \ \wedge\ S[t]\Downarrow \Rightarrow S[s]\Downarrow$. We restrict our attention here to $S[s]\Downarrow \Rightarrow S[t]\Downarrow$ because $S[t]\Downarrow \Rightarrow S[s]\Downarrow$ could be treated in a similar way. To prove $s \sim_c t$ we assume that $s \xrightarrow{T} t$ and $S[s]\Downarrow$ holds, i.e. there is a WHNF s', such that $S[s] \xrightarrow{no,k} s'$ (see Figure 3(a)). It remains to show that there also exists a sequence of normal order reductions from $S[t]$ to a WHNF. This can often be done by induction on the length k of the given normal order reduction $S[s] \xrightarrow{no,k} s'$ using *complete sets of reduction diagrams*. Therefore we split $S[s] \xrightarrow{no,k} s'$ into $S[s] \xrightarrow{no} s_o \xrightarrow{no,k-1} s'$ (see Figure 3(b)). Then an applicable *forking diagram* defines how the fork $s_0 \xleftarrow{no} S[s] \xrightarrow{T} S[t]$ can be closed specifying two sequences of transformations such that a common expression t' is eventually reached: one starting from $S[t]$ consisting only of no-reductions and one starting from s_0 consisting of some other reductions (that are not normal order) denoted by T' in Figure 3(c).

$$\begin{tikzcd} S[s] \arrow[r, "T"] \arrow[d, "no{,}k"'] & S[t] \\ s' & \end{tikzcd}$$

(a) Forking in the proof of $s \leq_c t$

$$\begin{tikzcd} S[s] \arrow[r, "T"] \arrow[d, "no"'] & S[t] \\ s_0 \arrow[d, "no{,}k-1"'] & \\ s' & \end{tikzcd}$$

(b) Splitting the no-sequence

$$\begin{tikzcd} S[s] \arrow[r, "T"] \arrow[d, "no"'] & S[t] \arrow[d, dashed, "no{,}*"'] \\ s_0 \arrow[r, dashed, "T'{,}*"'] \arrow[d, "no{,}k-1"'] & t' \\ s' & \end{tikzcd}$$

(c) Application of a forking diagram

$$\begin{tikzcd} S[s] \arrow[r, "T"] \arrow[d, "no"'] & S[t] \arrow[d, dashed, "no{,}*"'] \\ s_0 \arrow[r, dashed, "T'{,}*"'] \arrow[d, "no{,}k-1"'] & t' \arrow[d, dashed, "no{,}*"'] \\ s' \arrow[r, dashed, "T'{,}*"'] & t'' \ \text{WHNF} \end{tikzcd}$$

(d) Inductive proof of $s \leq_c t$

Figure 3: Sketch of the correctness proof for $s \xrightarrow{T} t$

A set of forking diagrams for a transformation T is *complete* if the set comprises an applicable diagram for every forking situation. If we have a complete set of forking diagrams we often can inductively construct a terminating reduction sequence for $S[t]$ if $S[s]\Downarrow$ (as indicated in Figure 3(d)). To prove $S[t]\Downarrow \Rightarrow S[s]\Downarrow$ another complete set of diagrams called *commuting diagrams* is required which usually can be deduced from a set of forking diagrams (see [15]). We restrict our attention to complete sets of forking diagrams.

Example 2.8 *Example forking diagrams are*

$$\begin{tikzcd}
\cdot \arrow[r, "i\mathcal{S},llet\text{-}e"] \arrow[d, "no,llet\text{-}in"'] & \cdot \arrow[d, dashed, "no,llet\text{-}in"] \\
\cdot \arrow[r, dashed, "i\mathcal{S},llet\text{-}e"] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "i\mathcal{S},llet\text{-}e"] \arrow[d, "no,llet\text{-}in"'] & \cdot \arrow[ldd, dashed, "no,llet\text{-}in"] \\
\cdot \arrow[d, dashed, "no,llet\text{-}e"'] & \\
\cdot &
\end{tikzcd}$$

where the dashed lines indicate existentially quantified reductions and the prefix $i\mathcal{S}$ *marks that the transformation is not a normal order reduction (but a so called* internal reduction *which we also call transformation), and occurs within a surface context. By application of the diagram a fork between a* (no,llet-e) *and the transformation* (llet-in) *can be closed. The forking diagrams specify two reduction sequences such that a common expression is eventually reached. The following reduction sequence illustrates an application of the above diagram:*

$$(\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s)\ \texttt{in}\ (\texttt{letrec}\ Env_3\ \texttt{in}\ r))$$

$$\xrightarrow{no,llet\text{-}in} \quad (\texttt{letrec}\ Env_1, Env_3, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s)\ \texttt{in}\ r)$$

$$\xrightarrow{i\mathcal{S}\vee no,llet\text{-}e} \quad (\texttt{letrec}\ Env_1, Env_3, Env_2, x = s\ \texttt{in}\ r)$$

the last reduction is either an no-reduction if $r = A[x]$, *otherwise it is an internal reduction*

$$\xrightarrow{i\mathcal{S},llet\text{-}e} \quad (\texttt{letrec}\ Env_1, Env_2, x = s\ \texttt{in}\ (\texttt{letrec}\ Env_3\ \texttt{in}\ r))$$

$$\xrightarrow{no,llet\text{-}in} \quad (\texttt{letrec}\ Env_1, Env_2, Env_3, x = s\ \texttt{in}\ r)$$

The square diagram covers the case, where (no,llet-in) *is followed by an internal reduction. The triangle diagram covers the other case, where the reduction following* (no,llet-in) *is* (no,llet-e). *One can view the forking diagram as a description of local confluence.*

The computation of a complete set of diagrams by hand is cumbersome and error-prone. Nevertheless the diagram sets are essential for proving correctness of a large set of program transformations in this setting. For this reason we are interested in automatic computation of complete diagram sets.

The first step in the computation of a complete set of forking diagrams for a transformation T is the determination of all forks of the form $\xleftarrow{no,red} \cdot \xrightarrow{i\mathcal{S},T}$ where *red* is an no-reduction and T is not a normal order reduction (but a transformation in an $\mathcal{S}$-context). Such forks are given by *overlaps* between no-reductions and the transformation. Informally we say that *red* and T overlap in an expression s if s contains a normal order redex *red* and a T redex (in a surface context). To find an overlap between an no-reduction *red* and a transformation T it is sufficient, by definition of the normal order reduction, to determine all surface-positions in *red* where a T-redex can occur. For the computation of all forks we have to consider only *critical overlaps* where an overlap does not occur at a variable position (Example 2.8 illustrates such a critical overlap). Forks stemming from non-critical overlaps at variable positions can always be closed by a predefined set of standard diagrams. All (critical) overlaps between no-reductions and a given transformation T can be computed by a variant of critical pair computation based on unification. The employed unification procedure will be explained in the next section.

3 Encoding Expressions as Terms in a Combination of Sorted Equational Theories and Context

In this section we develop a unification method to compute proper overlaps for forking diagrams. According to the context lemma for surface contexts (Lemma 2.7) we restrict the overlaps to the transformations applied in surface contexts. A complete description of a single overlap is the unification equation $S[l_{T,i}] \doteq l_{no,j}$, where $l_{T,i}$ is a left hand side in Figure 1, and $l_{no,j}$ a left hand side in Figure 2, and S means a surface context. To solve these unification problems we translate the meta-expressions from transformations and no-reduction rules into many sorted terms with some special constructs to mirror the syntax of the reduction rules in the lambda calculus. The constructs are i) context variables of different context classes $\mathcal{A}, \mathcal{S}$ and $\mathcal{C}$, ii) a left-commutative function symbol *env* to model that bindings in letrec-environments can be rearranged iii) a special construct $\mathtt{BCh}(\ldots)$ to represent binding chains of variable length as they occur in no-reduction rules.

The presented unification algorithm is applicable to terms with the mentioned extra constructs. We do not use the general unification combination algorithms in [11, 2], since we only have a special theory *LC* that models multi-sets of bindings in letrec-environments of our calculus, and moreover, it is not clear how to adapt the general combination method to context classes and binding chains.

3.1 Many Sorted Signatures, Terms and Contexts

Let $\mathcal{S} = \mathcal{S}_1 \uplus \mathcal{S}_2$ be the disjoint union of a set of theory-sorts $\mathcal{S}_1$ and a set of free sorts $\mathcal{S}_2$. We assume that *Exp* is a sort in $\mathcal{S}_2$. Let $\Sigma = \Sigma_1 \uplus \Sigma_2$ be a many-sorted signature of (theory- and free) function symbols, where every function symbol comes with a fixed arity and with a single sort-arity of the form $f : S_1 \times \ldots \times S_n \to S_{n+1}$, where S_i for $i = 1, \ldots, n$ are the argument-sorts and S_{n+1} is called resulting sort. For every $f \in \Sigma_i$ for $i = 1, 2$ the resulting sort must be in $\mathcal{S}_i$. Note, however, that there may be function symbols $f \in \Sigma_i$ that have argument-sorts from $\mathcal{S}_j$, for $i \neq j$. There is a set $\mathcal{V}^0$ of first-order variables that are 0-ary and have a fixed sort and are ranged over by $x, y, z, \ldots$, perhaps with indices. We write x^S if the variable x has the sort S. There is also a set $\mathcal{V}^1$ of context-variables which are unary and are ranged over by X, Y, Z, perhaps with indices. We assume that for every sort S, there is an infinite number of variables of this sort, and that there is an infinite number of context variables of sort $Exp \to Exp$. Let $\mathcal{V} = \mathcal{V}^0 \cup \mathcal{V}^1$. The set of terms $\mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V})$ is the set of terms built according to the grammar $x \mid f(t_1, \ldots, t_n) \mid X(t)$, where sort conditions are obeyed. Let $Var(t)$ be the set of first-order variables that occur in t and let $Var^1(t)$ be the set of context variables that occur in t. A context C is a term in $\mathcal{T}(Exp, \Sigma \cup [\cdot], \mathcal{V})$ such that there is exactly one occurrence of a the special hole constant $[\cdot]$ in the context and the sort at the position of the hole is *Exp*.

A term s without occurrences of variables is called *ground*. We also allow sorts without any ground term, also called *empty sorts*, since this is required in our encoding of bound variables. The term s is called *almost ground*, if for every variable x in s, there is no function symbol in Σ where the resulting sort is the sort of x, and hence no ground term of this sort.

A substitution σ is a mapping $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$, such that $\sigma(x^S)$ is a term of sort S and $\sigma(X)$ is a context. As usual we extend σ to terms, where every variable x in a term is replaced by $\sigma(x)$.

3.2 Encoding of L_{need}-Expressions as Terms

The sort and term structure according to the expression structure of the lambda calculus L_{need} (from section 2.1) is as follows. There are the following sorts: $Bind, Env, Exp, BV$, for bindings, environments, expressions and bound variables, respectively; where $\mathcal{S}_1 = \{Env\}$ and $\mathcal{S}_2 = \{Bind, Exp, BV\}$. There are the following function symbols:

theory function symbols (Σ_1)	free function symbols (Σ_2)	
$emptyEnv :: \quad Env$	$let :: \; Env \times Exp \to Exp$	$bind :: BV \times Exp \to Bind$
$env :: \quad Bind \times Env \to Env$	$app :: Exp \times Exp \to Exp$	$var :: BV \quad \to Exp$
	$lam :: BV \times Exp \to Exp$	

Note that there are free function symbols that map from *Env* to *Exp*, but there is no free function symbol that maps to *Env*. Note also that there is no function symbol with resulting sort *BV*, hence this is an empty sort, and every term of sort *BV* is a variable.

It is convenient to have a notation for nested *env*-expressions: $env^*(\{t_1,\ldots,t_m\} \cup r)$ denotes the term $env(t_1, env(t_2,\ldots, env(t_m, r)\ldots))$, where r is not of the form $env(s,t)$. Due to our assumptions on terms of sort *Env* and the sort of context variables, only the constant *emptyEnv* and variables are possible for r.

As an example the expression $(\texttt{letrec}\; x = \lambda y.y, z = x\,x \;\texttt{in}\; z)$ is encoded as $let(env^*(\{bind(x, lam(y, var(y))), bind(z, app(var(x), var(z)))\} \cup emptyEnv), var(z))$, where x,y,z are variables of sort BV.

To model the multi-set property of letrec-environments, i.e., that bindings can be reordered, we use the equational theory *left-commutativity* (*LC*) with the following axiom: $env(x, env(y,z)) = env(y, env(x,z))$ (for the *LC*-theory and unification modulo LC see [5, 4]). The equational theory *LC* is a congruence relation on the terms, which is denoted as $=_{LC}$. The *pure equational theory* is defined as restricted to the axiom-signature, i.e. to the terms $\mathcal{T}(\{Env, Bind\}, \Sigma_1, \mathcal{V}_{Env} \cup \mathcal{V}_{Bind})$, where $\mathcal{V}_S$ is the set of variables of sort S. The *combined equational theory* is defined on the set of terms $\mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$. Note that it is a disjoint combination w.r.t. the function symbols, but not w.r.t. the sorts.

The following facts about the theory *LC* can easily be verified:

Lemma 3.1 *For the equation theory LC, the following holds in $\mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$:*

- *The terms in the LC-axioms are built only from Σ_1-symbols and variables, and the axioms relate two terms of equal sort which must be in $\mathcal{S}_1$.*
- *For every equation $s =_{LC} t$, the equality $Var(s) = Var(t)$ holds.*
- *The equational theory LC is non-collapsing, i.e, there is no equation of the form $x =_{LC} t$, where t is not the variable x.*
- *If $C[s] =_{LC} t$ and s has a free function symbol as top symbol, then there is a context C' and a term s' such that $C[s] =_{LC} C'[s'], C' =_{LC} C$, $s =_{LC} s'$ and $C'[s'] = t$. This follows from general properties of combination of equational theories and properties of the theory LC.*
- *The equational theory LC has a finitary and decidable unification problem (see[5, 4]).*

In order to capture binding chains of variable length as they occur in the definition of the no-reduction rules (Figure 2) the syntax construct $\texttt{BCh}(N_1,N_2)$ is introduced, where N_i are integer variables that can be instantiated with $N_1 \mapsto n_1$, $N_2 \mapsto n_2$, where $0 < n_1 < n_2$. An instance $\texttt{BCh}(n_1,n_2)$ for $n_1,n_2 \geq 1$ represents the following binding chain: $bind(y_{n_1+1}, A_{n_1+1}(var(y_{n_1})))$, $bind(y_{n_1+2}, A_{n_1+2}(var(y_{n_1+1}))), \ldots, bind(y_{n_2}, A_{n_2}(var(y_{n_2-1})))$, where the names y_i, A_i are reserved for these purposes and are all distinct. The $\texttt{BCh}$-expressions are permitted only in the env^*-notation, like a sub-multi-set, and we denote this for example as $env^*(\ldots \cup \texttt{BCh}(N_1,N_2) \cup r)$.

Context-classes are required to correctly model the overlappings in L_{need}. The transformations in Figure 1 contain only C-contexts, whereas in Figure 2 there are also $\mathcal{A}$- and $\mathcal{R}$-contexts, and the overlapping also requires surface contexts $\mathcal{S}$. The grammar definition of $\mathcal{A}$-, $\mathcal{R}$- and $\mathcal{S}$-contexts (definition 2.3) justifies the replacement of $\mathcal{R}$-contexts by expressions containing only $\mathcal{A}$-contexts and $\texttt{BCh}$-expressions.

Thereby some rules of Figure 2 may be split into several rules. The context class $\mathcal{C}$ means all contexts and $\mathcal{S}$ means all contexts where the hole is not in an abstraction. In the term encoding, these translate to context variables. The unification algorithm must know how to deal with context variables of classes $\mathcal{A}$, $\mathcal{S}$ and $\mathcal{C}$. The partial order on context classes is $\mathcal{A} < \mathcal{S} < \mathcal{C}$. For every almost ground context C it can be decided whether C belongs to $\mathcal{A}$ (or $\mathcal{S}$, respectively). We will use the facts that equational deduction w.r.t. LC does not change the context class of almost ground contexts, and that prefix and suffix contexts of almost ground contexts C have the same context class as C (among $\mathcal{A}$, $\mathcal{S}$ and $\mathcal{C}$).

4 A Unification Algorithm LCSX for Left-Commutativity, Sorts and Context-Variables

We define unification problems and solutions as extension of equational unification (see [3]).

A *unification problem* is a pair (Γ, Δ), where $\Gamma = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$, the terms s_i and t_i are of the same sort for every i and may also contain `BCh`-expressions, every context variable is labelled with a context class symbol, and $\Delta = (\Delta_1, \Delta_2)$ is a constraint consisting of a set of context variables Δ_1 and a set Δ_2 of equations and inequations of the form $N_i + 1 = N_j$ and $N_i < N_j$ for the integer variables N_i. The intention is that Δ_1 consists of context variables that must not be instantiated by the empty context, and that the constraints Δ_2 hold for $\sigma(N_i)$ after instantiating with σ.

A *solution* σ of (Γ, Δ), with $\Gamma = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$ is a substitution σ according to the following conditions: i) it instantiates variables by terms, context variables by contexts of the correct context class that are nontrivial if contained in Δ_1, and the integer variables N_i by positive integers according to the constraint Δ_2. ii) $\sigma(s_i), \sigma(t_i)$ are almost ground for all i. It is assumed that the `BCh`-constructs `BCh`(n_1, n_2) are expanded into a binding chain as explained above, iii) $\sigma(s_i) =_{LC} \sigma(t_i)$ for all i.

A unification problem Γ is called *almost linear*, if every context variable occurs at most once and every variable of a non-empty sort occurs at most once in the equations.

Definition 4.1 *Let Π_T be the set of left hand sides of reduction rules from Figure 1 and Π_{no} the set of left hand sides of no-reduction rules from Figure 2 where the reduction contexts R in (lbeta) and (lapp) are instantiated by the four possibilities for R: A,* (`letrec` Env `in` A), (`letrec` $y_1 = A, Env$ `in` $A_2[y_1]$), (`letrec` $y_{N_1} = A, BCh(N_1, N_2), Env$ `in` $A[y_{N_2}]$) *with constraint $N_1 < N_2$. The meta-variable s in the cp rules (that can be either a variable or an abstraction) is instantiated by $var(z)$ and an abstraction $\lambda x.t$ where t denotes a meta-variable for an arbitrary expression. With Π'_T, Π'_{no} we denote the sets where left hand sides of rules are encoded as terms.*

We consider the set of unification problems $\Gamma_i = \{S(l_{T,i}) \doteq l_{no,j} \mid l_{no,j} \in \Pi'_{no}\}$ with $l_{T,i} \in \Pi'_T$ and S is a surface context variable. The sets Π'_T and Π'_{no} are assumed to be variable disjoint, which can be achieved by renaming. The initial set Δ_1 of context variables only contains the A_2-context from the (cp-e)-reductions, and Δ_2 may contain some initial constraints from the rules. The pairs (Γ_i, Δ) are called the initial L_{need}-forking-problems.

Note that initial L_{need}-forking-problems are almost linear, there is at most one `BCh`-construct, which is in the environment of the topmost let-expression, and there are no variables of type *Bind*.

Definition 4.2 *A* final unification problem *S of an initial Γ is a set of equations $s_1 \doteq t_1, \ldots, s_n \doteq t_n$, such that $S = S_{BV} \cup S_{\neg BV}$, and every equation in S_{BV} is of the form $x \doteq y$ where x, y are of sort BV and every equation in $S_{\neg BV}$ is of the form $x \doteq t$, where x is not of sort BV, and the equations in $S_{\neg BV}$ are in DAG-solved form.*

Given a final unification problem S, the represented solutions σ could be derived by first instantiating the integer variables, expanding the BCh-constructs into binding chains, instantiating all context variables and variables that are not of sort BV and then turning the equations into substitutions.

A final unification problem S derived from Γ *satisfies the distinct variable convention* (DVC), if for every derived solution σ, all terms in $\sigma(\Gamma)$ satisfy the DVC. This property is decidable: If $t_1 \doteq t_2$ is the initial problem, then apply the substitution σ derived from S to t_1. The DVC is violated if the following condition holds: Let M_{BV} be the set of BV-variables occurring in $\sigma(t_1)$. For every BCh-construct $\mathtt{BCh}(N_1,N_2)$ occurring in $\sigma(t_1)$ we add the variable y_{N_2} to M_{BV}. If $\sigma(t_1)$ makes two variables in M_{BV} equal, then the DVC is violated, and the corresponding final problem is discarded.

Example 4.3 *Unifying (the first-order encodings of) $\lambda x.\lambda y.x$ and $\lambda u.\lambda v.v$, the unification succeeds and generates an instance that represents $\lambda x.\lambda x.x$, which does not satisfy the DVC. Thus a variant of our unification can efficiently check alpha-equivalence of lambda-expressions that satisfy the DVC.*

We proceed by describing a unification algorithm starting with initial L_{need}-unification problems (Γ,Δ). It is intended to be complete for all common instances that represent L_{need}-expressions that satisfy the DVC, i.e. where all bound variables are distinct and the bound variables are distinct from free variables. Final unification problems that lead to expressions that do not satisfy the DVC are discarded.

Given an initial unification problem $\Gamma = \{s_1 \doteq t_1\};\Delta$, the (non-deterministic) unification algorithm described below will non-deterministically compute a final unification problem S or fail. A finite complete set of final unification problems can be attained by gathering all final unification problems in the whole tree of all non-deterministic choices. We implicitly use symmetry of $\doteq$ if not stated otherwise. We divide Γ in a solved part S, (a final unification problem), and a still to be solved part P. We usually omit Δ in the notation if it is not changed by the rule.

Standard unification rules.

Dec $\dfrac{S;\ \{f(s_1,\dots,s_n) \doteq f(t_1,\dots,t_n)\} \uplus P}{S;\ \{s_1 \doteq t_1,\dots,s_n \doteq t_n\} \cup P}$ If f is a free function symbol (i.e. $f \neq env$).

Solve $\dfrac{S;\ \{x \doteq t\} \uplus P}{\{x \doteq t\} \cup S;\ P}$ **Trivial** $\dfrac{S;\ \{s \doteq s\} \uplus P}{S;\ P}$

Fail $\dfrac{S;\ \{f(\dots) \doteq g(\dots)\} \uplus P}{Fail}$ **DVC-Fail** $\dfrac{S;\ \emptyset}{Fail}$ If S is final and the DVC is violated w.r.t. the initial problem.

Note that the occurs-check is not necessary, since P is almost linear and an equation $x \doteq t$ for variables x of type BV implies that t is a variable.

Solving equations with context variables. The rules for terms with contexts as top symbol using their context classes are as follows: The following rule operates on context variables at any position:

Empty-C $\dfrac{S;\ P;\ \Delta_1}{\begin{array}{l}\text{select one of the following possibilities}\\ S;\ P;\ \{X\}\cup\Delta_1 \quad \text{or} \quad \{X \mapsto [\cdot]\}\cup S;\ \{X \mapsto [\cdot]\}P;\ \Delta_1\end{array}}$ If X occurs in P and $X \notin \Delta_1$.

Assume there is an equation $X(s) \doteq t$, where the top symbol of t is not a context variable and $X \in \Delta_1$. Note that the sort of $X(s)$ is Exp. There are the following possibilities:

Dec-CA $\dfrac{S;\ \{X(s) \doteq app(t_1,t_2)\} \uplus P}{\{X \mapsto app(X',t_2)\} \cup S;\ \{X'(s) \doteq t_1\} \cup P}$

X' is a fresh context variable of the same context class as X.

Dec-CC $\dfrac{S;\ \{X(s) \doteq f(t_1,t_2)\} \uplus P}{\{X \mapsto f(t_1,X')\} \cup S;\ \{X'(s) \doteq t_2\} \cup P}$ if t_2 is of sort *Exp*.

X' is a fresh context variable of the same context class as X (it may only be $\mathcal{C}$ or $\mathcal{S}$) and f is a function symbol such that $f \in \{let, app\}$.

Dec-CL $\dfrac{S;\ \{X(s) \doteq let(t_1,t_2)\} \uplus P}{\{X \mapsto let(env^*(\{bind(x,X')\} \cup z),t_2)\} \cup S;\ \{env^*(\{bind(x,X'(s))\} \cup z) \doteq t_1\} \cup P}$

If X is of context class $\mathcal{S}$ or $\mathcal{C}$. X' is a fresh context variable of the same context class as X.

Dec-Lam $\dfrac{S;\ \{X(s) \doteq lam(t_1,t_2)\} \uplus P}{\{X \mapsto lam(t_1,X')\} \cup S;\ \{X'(s) \doteq t_2\} \cup P}$

If X is of class $\mathcal{C}$. X' is a fresh context variable of the class $\mathcal{C}$.

Fail-Lam $\dfrac{S;\ \{X(s) \doteq lam(t_1,t_2)\} \uplus P}{Fail}$ **Fail-Var** $\dfrac{S;\ \{X(s) \doteq var(x)\} \uplus P}{Fail}$

If X is of class $\mathcal{A}$ or $\mathcal{S}$.

Given an equation $X(s) \doteq Y(t)$, with $X,Y \in \Delta_1$, let $\mathcal{D}$ be the smaller one of the context classes of X,Y. Then select one of the following possibilities:

Merge-P $\dfrac{S;\ \{X(s) \doteq Y(t)\} \uplus P}{\{Y \mapsto ZY', X \mapsto Z\} \cup S;\ \{s \doteq Y'(t)\} \cup P}$

Y' is a fresh context variable of the same context class as Y, and Z has context class $\mathcal{D}$.

Merge-FA $\dfrac{S;\ \{X(s) \doteq Y(t)\} \uplus P}{\{X \mapsto Z(app(X',Y'(t))), Y \mapsto Z(app(X'(s),Y'))\} \cup S;\ P}$

If exactly one of the context classes of X,Y is $\mathcal{A}$. W.l.o.g. let X be of context class $\mathcal{A}$. X',Y' are fresh context variables of the same context class as X,Y, respectively, and Z is a fresh context variable of context class $\mathcal{A}$.

Merge-FC

$$S;\ \{X(s) \doteq Y(t)\} \uplus P$$

choose either of the following possibilities

$\{X \mapsto Z(app(X',Y'(t))), Y \mapsto Z(app(X'(s),Y'))\} \cup S;\ P$

$\{X \mapsto Z(let(env^*(\{bind(x,X')\} \cup, z),Y'(t))), Y \mapsto Z(let(env^*(\{bind(x,X'(s))\} \cup z),Y'))\} \cup S;\ P$

$\{X \mapsto Z(let(env^*(\{bind(x,X'),bind(y,Y'(t))\} \cup z),w)),$
$\quad Y \mapsto Z(let(env^*(\{bind(x,X'(s)),bind(y,Y')\} \cup z),w))\} \cup S;\ P$

If the context classes of X,Y are different from $\mathcal{A}$. X',Y' are fresh context variables of the same context class as X,Y, respectively and Z is a fresh context variable of context class $\mathcal{D}$. The variables w,x,y,z are also fresh and of the appropriate sort.

Rules for Multi-Set Equations. The following additional (non-deterministic) unification rules are sufficient to solve nontrivial equations of type *Env*, i.e. proper multi-set-equations, which must be of the form $env^*(L_1 \cup r_1) \doteq env^*(L_2 \cup r_2)$, where r_1, r_2 are variables or the constant *emptyEnv*. We will use the notation L for sub-lists in env^*-expressions and the notation $L_1 \cup L_2$ for union. In the terms $env^*(L \cup t)$, we assume that t is not of the form $env(\ldots)$. It is also not of the form $X(\ldots)$ due to the sort assumptions. Other free function symbols are disallowed, hence t can only be a variable or the constant *emptyEnv*. The components in the multi-set may be expressions of type *Bind*, i.e., variables or expressions with top symbol *bind*, or a `BCh`$(\ldots)$-component that represents several terms of type *Bind*. We also use the convention that in the conclusions of the rules an empty environment $env^*(\{\ \} \cup r)$ without any bindings and just a variable r is identified with r. Note that the lists allow multi-set operations like reorderings.

Due to the initial encoding of reduction rules, if a $\mathtt{BCh}(N_1,N_2)$-construct occurs in a term in P, it occurs in an env^*-list, hence there is also a binding $y_{N_1} = s$ in the env^*-list, and the list is terminated with a variable derived from the environment-variable Env. In equations, the $\mathtt{BCh}(\ldots)$-components initially appear only on one side, which cannot be changed by the unification. Also the env^*-list is an immediate sub-term of a top let-expression, which may change after applying unification rules. Due to these conditions, we assume that the left term in the equation does not contain $\mathtt{BCh}(\ldots)$-components.

If there is an equation $env^*(L_1 \cup r_1) \doteq env^*(L_2 \cup r_2)$, then select one of the following possibilities:

Solve-E $$\frac{S;\ \{env^*(L_1 \cup r_1) \doteq env^*(L_2 \cup r_2)\} \uplus P}{\{r_1 \mapsto env^*(L_2 \cup z_3), r_2 \mapsto env^*(L_1 \cup z_3)\} \cup S;\ P}$$ If r_1, r_2 are variables; z_3 is a fresh variable.

Dec-E $$\frac{S;\ \{env^*(L_1 \cup r_1) \doteq env^*(L_2 \cup r_2)\} \uplus P}{S;\ \{t_1 \doteq t_2, env^*(L_1 \setminus \{t_1\} \cup r_1) \doteq env^*(L_2 \setminus \{t_2\} \cup r_2)\} \uplus P}$$
If L_1 and L_2 contain binding expressions t_1 and, t_2.

Dec-Ch
$$\frac{S;\ \{env^*(L_1 \cup r_1) \doteq env^*(\mathtt{BCh}(N_1,N_2) \cup L_2 \cup r_2)\} \uplus P;\ (\Delta_1, \{N_1 < N_2\} \cup \Delta_2)}{\begin{array}{l}\text{select one of the following possibilities}\\ (i)\ S;\ \{t_1 \doteq bind(y_{N_2}, A_{N_2}(var(y_{N_1}))),\\ \quad env^*(L_1 \setminus \{t_1\} \cup r_1) \doteq env^*(L_2 \cup r_2)\} \cup P;\ \{A_{N_2}\} \cup \Delta_1, \{N_1 + 1 = N_2\} \cup \Delta_2\\ (ii)\ S;\ \{t_1 \doteq bind(y_{N_3}, A_{N_3}(var(y_{N_1}))),\\ \quad env^*(L_1 \setminus \{t_1\} \cup r_1) \doteq env^*(\mathtt{BCh}(N_3,N_2) \cup L_2 \cup r_2)\} \cup P;\ \{A_{N_3}\} \cup \Delta_1, \{N_1 + 1 = N_3, N_3 < N_2\} \cup \Delta_2\\ (iii)\ S;\ \{t_1 \doteq bind(y_{N_2}, A_{N_2}(var(y_{N_3}))),\\ \quad env^*(L_1 \setminus \{t_1\} \cup r_1) \doteq env^*(\mathtt{BCh}(N_1,N_3) \cup L_2 \cup r_2)\} \cup P;\ \{A_{N_2}\} \cup \Delta_1, \{N_1 < N_3, N_3 + 1 = N_2\} \cup \Delta_2\\ (iv)\ S;\ \{t_1 \doteq bind(y_{N_4}, A_{N_4}(var(y_{N_3}))),\\ \quad env^*(L_1 \setminus \{t_1\} \cup r_1) \doteq env^*(\mathtt{BCh}(N_1,N_3) \cup \mathtt{BCh}(N_4,N_2) \cup L_2 \cup r_2)\} \cup P;\\ \quad \{A_{N_4}\} \cup \Delta_1, \{N_1 < N_3, N_3 + 1 = N_4, N_4 < N_2\} \cup \Delta_2\\ \text{Where } y_{N_2}, y_{N_3}, y_{N_4}, A_{N_2}, A_{N_3}, A_{N_4}, N_3, N_4 \text{ are fresh variables of appropriate sort.}\end{array}}$$

Fail-E $$\frac{S;\ \{env^*(L \cup t) \doteq emptyEnv\} \uplus P}{Fail}.$$
If L is nonempty, i.e contains at least one binding or at least one $\mathtt{BCh}$-expression.

An invariant of the rules that deal with $\mathtt{BCh}$ is that the variables N_i may appear at most twice in Γ; at most twice explicit in Δ_2 and at most once in $\mathtt{BCh}$-expressions.

4.1 Properties of the LCSX-Unification Algorithm

Lemma 4.4 *For initial problems, the algorithm LCSX terminates.*

Proof. For this we can ignore the rules that change Δ.

The following measure is used, which is a lexicographical combination of several component measures: μ_1 is the number of occurrences of *let* in P; the second component μ_2 is the following size-measure, where $env^*(L \cup r)$ has measure $7m + m' + \sum \mu_2(t_i) + \mu_2(r)$ where m is the number of *bind*-expressions in L and and m' is the number of $\mathtt{BCh}$-expressions in L.

The critical applications are the guessing rules for equations with top-context variables, and the rules for multi-equations. The context variable-guessing either decreases the size or the number of occurrences of let. The multi-equation rules in rule **Dec-Ch** have to be analyzed. The new constructed bind-term has size 5, so the subcases (i) – (iii) strictly reduce the size. The subcase (iv) adds 6 to the size due to new sub-terms, and removes 7 since t_1 is a non-BCh-expression and removed from the multi-set. □

Lemma 4.5 *The non-deterministic rule-based unification algorithm LCSX is sound and complete in the following sense: every computed final unification problem that leads to an expression satisfying the DVC represents a set of solutions and every solution of the initial unification problem that represents an expression satisfying the DVC is represented by one final system of equations.*

Proof. Soundness can be proved by standard methods, since rules are either instantiations or instantiations using the theory LC.

Completeness can be proved, if every rule is shown to be complete, and if there are no stuck unification problems that have solutions. The **Solve** rules are complete since solved variables (in equations of the form $x \doteq t$) are just marked as such, i.e. moved to a set of solved equations. Solving equations $X(s) \doteq t$ is complete: if t is a variable, then it can be replaced; if t is a proper term of type *Exp*, then all cases are covered by the rules. In the case that the equation is $X(s) = Y(t)$, the rules are also complete, and also respect the context classes of X,Y. If the equation is $s \doteq s$, then it will be removed, and if it is of the form $f(\ldots) \doteq f(\ldots)$ then decomposition applies. In the case that the top symbol is *env*, the rules for multi-equations apply, i.e., the rules for env^*. Using the properties of the equational theory *LC* and the considerations in [5]), we see that the rules are complete. □

Theorem 4.6 *The rule-based algorithm LCSX terminates if applied to initial L_{need}-forking-problems. Thus it decides unifiability of these sets of equations. Since it is sound and complete, and the forking possibilities are finite, the algorithm also computes a finite and complete set of final unification problems by gathering all possible results.*

Theorem 4.7 *The computation of all overlaps between the rules in Figure 1 and left hand sides of normal order reductions in Figure 2 can be done using the algorithm LCSX. The unification algorithm terminates in all of these cases and computes a finite set of final unification problems and hence all the critical pairs w.r.t. our normal order reduction.*

5 Running the Unification Algorithm LCSX

Example 5.1 *The goal is to compute a complete set of forks for the transformation (cp-e)*

$$(\texttt{letrec}\ x = s, Env, z = C[x]\ \texttt{in}\ r) \to (\texttt{letrec}\ x = s, Env, z = C[s]\ \texttt{in}\ r)$$

from Figure 1. We instantiate the meta-variable s by the expression $\lambda w.t$ and translate the left hand side of the rule into the term language, resulting in the following initial forking problem to be solved

$$\{S(let(env^*(\{bind(x,lam(w,t)),bind(z,C(var(x)))\} \cup Env),r)) \doteq l_{no,j}\}.$$

where $l_{no,j}$ is an encoded left hand side of an no-reduction rule. We pick a single equation from this set:

$$\begin{aligned} & S(let(env^*(\{bind(x,lam(w,t)),bind(z,C(var(x)))\} \cup Env_1),r)) \\ \doteq\ & let(env^*(\{bind(x',lam(w',t')),bind(y_{N_1},A_{N_1}(var(x')))\} \cup BCh(N_1,N_2) \cup Env_2),A(y_{N_2})) \end{aligned}$$

*which describes the overlaps between the (cp-e) transformation and the normal order (cp-e-c) reduction. No we compute one possible final problem via the presented unification algorithm. A nontrivial possibility is to choose $S = [\cdot]$ via the **Empty-C**-rule and then using decomposition for let which leads to $r = A(y_{N_2}))$ and the equation*

$$\begin{array}{rl} & env^*(\{bind(x,lam(w,t)),bind(z,C(var(x)))\} \cup Env_1) \\ \doteq & env^*(\{bind(x',lam(w',t')),bind(y_{N_1},A_{N_1}(var(x')))\} \cup \mathtt{BCh}(N_1,N_2) \cup Env_2). \end{array}$$

*One choice for the next step (via the rule **Dec-Ch**) results in the equations:*

$$\begin{array}{rl} & bind(z,C(var(x)) \doteq bind(y_{N_4},A_{N_4}(var(y_{N_3}))), \quad env^*(\{bind(x,lam(w,t))\} \cup Env_1) \\ \doteq & env^*(\{bind(x',lam(w',t')),bind(y_{N_1},A_{N_1}(var(x')))\} \cup (\mathtt{BCh}(N_1,N_3) \cup \mathtt{BCh}(N_4,N_2) \cup Env_2) \end{array}$$

*where one binding is taken from the $\mathtt{BCh}(N_1,N_2)$-construct and the chain is split around this binding into two remaining chains. The two bindings $bind(x,lam(w,t))$ and $bind(x',lam(w',t'))$ are unified (via **Dec-E**) and then we solve the equation between the environments (**Solve-E**) and (after three additional **Dec**-steps, two for bind and one for lam) we arrive at the system*

$$\begin{array}{l} C(var(x)) \doteq A_{N_4}(var(y_{N_3})), z \doteq y_{N_4}, x \doteq x', w \doteq w', t \doteq t', Env_2 \doteq env^*(\{bind(x,lam(w,t))\} \cup Env_3), \\ Env_1 \doteq env^*(\{bind(x',lam(w',t')),bind(y_{N_1},A_{N_1}(var(x')))\} \cup \mathtt{BCh}(N_1,N_3) \cup \mathtt{BCh}(N_4,N_2) \cup Env_3). \end{array}$$

*Next we apply **Merge-FA** to the first equation, yielding*

$$C \doteq Z(app(A'_{N_2}(var(y_{N_3})),C')), A_{N_2} \doteq Z(app(A'_{N_2},C'(var(x))))$$

where Z, A'_{N_2} are of context class $\mathcal{A}$ and C' is of context class $\mathcal{C}$. The final representation is:

$$\begin{array}{l} S_{\neg BV} = \{S \doteq [\cdot], r \doteq A(y_{N_2}), C \doteq Z(\ldots), t \doteq t', A_{N_2} \doteq Z(\ldots), Env_2 \doteq env^*(\ldots), Env_1 \doteq env^*(\ldots)\} \\ S_{BV} = \{z \doteq y_{N_4}, x \doteq x', w \doteq w'\} \end{array}$$

The resulting expression is:

$$\begin{array}{l} \mathtt{letrec}\, x' = (\lambda w'.t'), y_{N_1} = A_{N_1}[x'], \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_1}^{N_3}, \\ \qquad y_{N_4} = Z[app(A'_{N_2}(var(y_{N_3})),C'[x'])], \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_4}^{N_2}, Env_2 \ \mathtt{in}\ A[y_{N_2}] \end{array}$$

The corresponding fork is given by reducing the expression with (no,cp-e-c) and (cp-e) respectively

$$\begin{tikzcd}
\begin{array}{c} \mathtt{letrec}\, x' = (\lambda w'.t'), y_{N_1} = A_{N_1}[x'], \\ \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_1}^{N_3}, y_{N_4} = Z[app(A'_{N_2}[var(y_{N_3})],C'[x'])], \\ \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_4}^{N_2}, Env_2 \ \mathtt{in}\ A[y_{N_2}] \end{array} \arrow[d, "no{,}cp-e-c"'] \arrow[dr, "i\mathcal{S}{,}cp-e"] & \\
\begin{array}{c} \mathtt{letrec}\, x' = (\lambda w'.t'), y_{N_1} = A_{N_1}[\lambda \mathbf{w}'.\mathbf{t}'], \\ \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_1}^{N_3}, y_{N_4} = Z[app(A'_{N_2}[var(y_{N_3})],C'[x'])], \\ \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_4}^{N_2}, Env_2 \ \mathtt{in}\ A[y_{N_2}] \end{array} & \begin{array}{c} \mathtt{letrec}\, x' = (\lambda w'.t'), y_{N_1} = A_{N_1}[x'], \\ \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_1}^{N_3}, \\ y_{N_4} = Z[app(A'_{N_2}[var(y_{N_3})],C'[\lambda \mathbf{w}'.\mathbf{t}'])], \\ \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_4}^{N_2}, Env_2 \ \mathtt{in}\ A[y_{N_2}] \end{array}
\end{tikzcd}$$

This fork can be closed by the sequence $\xrightarrow{i\mathcal{S},cp-e} \cdot \xleftarrow{no,cp-e-c}$. Notice that for the determination of all forks it is sufficient to compute final systems. The (possibly infinite) set of ground solutions is not required.

We implemented the presented unification algorithm LCSX in Haskell to compute all forks between transformations and no-reductions. The program computes 1214 overlaps for the L_{need} calculus, and also searches for closing reduction sequences. Via this method we were able to close (almost[1]) all forks. The complete sets of forking diagrams for the transformations llet and cp is in Figure 4 The implementation is available at: `http://www.ki.informatik.uni-frankfurt.de/research/dfg-diagram/en`. More informaiton can be found in [10].

[1]Some simple commuting diagrams for cp reductions are not automatically closed, due to renaming of bound variables.

$$\begin{tikzcd}
\cdot \arrow[r, "iS{,}lletin"] \arrow[d, "no{,}a"'] & \cdot \arrow[d, dashed, "no{,}a"] \\
\cdot \arrow[r, dashed, "S{,}lletin"'] & \cdot
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[r, "iS{,}lletin"] \arrow[d, "no{,}llete"'] & \cdot \arrow[ddl, dashed, "no{,}llete"] \\
\cdot \arrow[d, dashed, "no{,}llete"'] & \\
\cdot &
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[rr, "iS{,}lletin"] \arrow[d, dashed, "no{,}lapp"'] & & \cdot \arrow[d, dashed, "no{,}lapp"] \\
\cdot \arrow[r, dashed, "S{,}lapp"'] & \cdot \arrow[r, dashed, "S{,}lletin"'] & \cdot
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[r, "iS{,}llete"] \arrow[d, "no{,}a"'] & \cdot \arrow[d, dashed, "no{,}a"] \\
\cdot \arrow[r, dashed, "S{,}llete"'] & \cdot
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[r, "iS{,}cpin"] \arrow[d, "no{,}a"'] & \cdot \arrow[d, dashed, "no{,}a"] \\
\cdot \arrow[r, dashed, "S{,}cpin"'] & \cdot
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[r, "iS{,}cpe"] \arrow[d, "no{,}a"'] & \cdot \arrow[d, dashed, "no{,}a"] \\
\cdot \arrow[r, dashed, "S{,}cpe"'] & \cdot
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[rr, "iS{,}cpe"] \arrow[d, "no{,}cp"'] & & \cdot \arrow[d, dashed, "no{,}cp"] \\
\cdot \arrow[r, dashed, "S{,}cp"'] & \cdot \arrow[r, dashed, "S{,}cp"'] & \cdot
\end{tikzcd}$$

Figure 4: Complet sets of forking diagrams for llet and cp transformations.

6 Conclusion and Further Work

We have provided an method using first-order unification with equational theories, sorts, context variables and context classes and binding chains of variable length to compute all critical overlaps between a set of transformation rules and a set of normal order rules in a call-by-need lambda calculus with letrec-environments. Further work is to apply this method to further transformations and also to extend the method in order to make it applicable to other program calculi as in [15], where variable-variable bindings are present in the rules, and to calculi with data structures and case-expressions.

References

[1] Zena M. Ariola & Matthias Felleisen (1997): *The call-by-need lambda calculus*. J. Funct. Program. 7(3), pp. 265–301.

[2] Franz Baader & Klaus U. Schulz (1992): *Unification in the union of disjoint equational theories: Combining decision procedures*. In: Proc. of 11th CADE, LNCS 607, Springer, pp. 50–65.

[3] Franz Baader & Wayne Snyder (2001): *Unification Theory*. In: J. A. Robinson & A.Voronkov, editors: Handbook of Automated Reasoning, Elsevier and MIT Press, pp. 445–532.

[4] Evgeny Dantsin & Andrei Voronkov (1999): *A Nondeterministic Polynomial-Time Unification Algorithm for Bags, Sets and Trees*. In: Proc. of 2nd FoSSaCS, LNCS 1578, Springer, pp. 180–196.

[5] Agostino Dovier, Enrico Pontelli & Gianfranco Rossi (2006): *Set unification*. TPLP 6(6), pp. 645–701.

[6] Matthias Felleisen & Robert Hieb (1992): *The Revised Report on the Syntactic Theories of Sequential Control and State*. Theor. Comput. Sci. 103(2), pp. 235–271.

[7] Douglas J. Howe (1989): *Equality In Lazy Computation Systems*. In: Proc. of 4th LICS, pp. 198–203.

[8] D. E. Knuth & P. B. Bendix (1970): *Simple word problems in universal algebra*. In: J. Leech, editor: Computational problems in abstract algebra, Pergamon Press, pp. 263–297.

[9] Robin Milner (1977): *Fully abstract models of typed lambda-calculi*. Theor. Comput. Sci. 4(1), pp. 1–22.

[10] Conrad Rau & Manfred Schmidt-Schauß (2010): *Towards Correctness of Program Transformations Through Unification and Critical Pair Computation*. Frank report 41, Goethe-Universität, FB 12.

[11] Manfred Schmidt-Schauß (1989): *Unification in a Combination of Arbitrary Disjoint Equational Theories*. J. Symb. Comput. 8(1/2), pp. 51–99.

[12] Manfred Schmidt-Schauß (2007): *Correctness of Copy in Calculi with Letrec*. In: Proc. of 18th RTA, LNCS 4533, Springer, pp. 329–343.

[13] Manfred Schmidt-Schauß & David Sabel (2010): *On generic context lemmas for higher-order calculi with sharing*. Theor. Comput. Sci. 411(11-13), pp. 1521–1541.

[14] Manfred Schmidt-Schauß, David Sabel & Elena Machkasova (2010): *Simulation in the Call-by-Need Lambda-Calculus with letrec*. In: Proc. of 21th RTA, LIPIcs 6, pp. 295–310.

[15] Manfred Schmidt-Schauß, David Sabel & Marko Schütz (2008): *Safety of Nöcker's strictness analysis*. J. Funct. Program. 18(4), pp. 503–551.